Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт космических и информационных технологий
институт
Информатики
кафедра

УТВЕРЖДАЮ
Заведующий кафедрой
_____ А.С., Кузнецов
подпись инициалы, фамилия
« 13 » 06 2017 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

«Инструментальная поддержка поиска клонов в программном коде»
тема

09.04.03 «Программная инженерия»
код и наименование направления

«Программное обеспечение вычислительной техники и автоматизированных систем»
код и наименование магистерской программы

Научный руководитель _____ 5.06.17 доцент, к.т.н. А.С., Кузнецов
подпись, дата должность, ученая степень инициалы, фамилия

Выпускник _____ 5.06.17 А.А., Бабкина
подпись, дата инициалы, фамилия

Рецензент _____ 5.06.17 к.т.н. Д.В., Личаргин
подпись, дата должность, ученая степень инициалы, фамилия

Красноярск 2017

Реферат

Диссертация на соискание степени магистра по направлению программной инженерии на тему «Инструментальная поддержка поиска клонов в программном коде» включает в себя 65 страниц текстового документа, 10 рисунков, из них 4 блок-схемы и 3 блока псевдокода, в которых отражено архитектурное решение реализуемой программы, 26 использованных источников.

КЛОНЫ КОДА, ВЕКТОРНЫЙ АНАЛИЗ, ВЕКТОРНАЯ МОДЕЛЬ СЕМАНТИКИ, ТРЕХАДРЕСНЫЙ КОД, THREE-ADDRESS CODE, VECTOR-SPACE MODEL, COPY-PASTE PROGRAMMING, CPP, VSM.

Актуальность работы заключается в необходимости создания инструментального средства, способного отыскивать семантически схожие объекты программного кода.

Объектом исследования являются клоны трехадресного кода и методы, применяемые для их поиска. Предмет исследования – возможность применения, процесс создания и реализации векторной модели семантики для поиска клонов в трехадресном коде.

Целью данной работы является разработка приложения, основанного на векторной модели семантики и способного отыскать клоны трехадресного кода всех трех типов, возникающих при применении подхода копирования-вставки и модификации исходного кода.

Для достижения данной цели были поставлены и реализованы следующие **задачи**:

1. Провести обзор существующих инструментальных средств, изучить алгоритмы их работы, выявить положительные и отрицательные качества программного продукта.

2. Изучить алгоритм, основанный на векторной модели семантики, проверить его эффективность и применимость для поиска клонов в трехадресном коде, адаптировать данный алгоритм для поиска клонов трехадресного кода.

3. Реализовать приложение на основе предложенного алгоритма, проверить правильность и эффективность его работы.

При выполнении данной работы использовались следующие методы:

- метод векторной модели семантики;
- методы разделения графа на слабосвязанные подграфы;
- метод определения схожести, основанный на косинусной мере.

Научная новизна работы заключается в обосновании применимости и построении векторной модели семантики для поиска клонов трехадресного кода, разработке модели построения матрицы векторной модели семантики, с учетом специфики трехадресного кода.

В процессе написания работы была реализована программа поиска клонов трехадресного кода, основанная на векторной модели семантики. Разработанная программа может применяться для анализа как небольших и средних по количеству строк кода проектов, так и в крупномасштабных проектах.

Результаты применимости векторной модели семантики для поиска клонов в трехадресном коде, были представлены на двух научных конференциях, в виде статей [3]. По завершении конференций были изданы сборники трудов, входящие в РИНЦ.

Оглавление

ВВЕДЕНИЕ

В настоящее время существует большое множество паттернов (design patterns) и методов для облегчения работы программиста. Шаблоны проектирования представляют собой общее архитектурное решение какой – то часто возникающей проблемы в рамках одного проекта [24].

Помимо полезных паттернов, действительно помогающих при написании программ, также существуют и антипаттерны (anti – patterns) – шаблоны, применение которых на первый взгляд может показаться полезным. На практике их использование приводит к появлению большого количества ошибок в коде, код становится малопонятным и нечитабельным. Антипаттерны, в отличие от шаблонов проектирования, представляют собой неправильное решение проблемы [20].

По классификации Уильяма Брауна с дополнениями Нейла и Лапланте, антипаттерны делятся на 4 типа [14, 15, 16]:

- антипаттерны разработки (development anti – patterns) – технические проблемы и решения, с которыми связаны программисты;

- архитектурные антипаттерны (architectural anti – patterns) – проблемы, связанные со структурой системы;

- организационные антипаттерны (managerial anti – patterns) – проблемы, за решение которых отвечают менеджеры или группы менеджеров;

- антипаттерны среды (environmental antipatterns) - проблемы, возникшие по вине доминирующей в организации структуры и социальной моделью, являющейся результатом действующей в организации общественной политики [14].

Один из методологических антипаттернов, относящийся к антипаттернам разработки, является программирование методом

копирования-вставки. Программирование методом копирования и модификации существующих фрагментов кода приводит к тому, что на этапе тестирования возникают ошибки, которые трудно отыскать во множестве схожих участков кода. В итоге программисту требуется дополнительное время на отыскание и устранение ошибок, приведение кода в более понятный вид. Более подробно проблемы связанные с применением данного подхода, описаны в разделе 1.3.

Вместо того чтобы реализовать общее решение для схожих по структуре проблем, программисты довольно часто применяют copy-paste подход, который порождает большое количество схожих по структуре и функционалу фрагментов кода. Такие фрагменты принято называть клонами программного кода. По степени схожести фрагментов кода можно выделить три основных типа клонов (полные определения даны в разделе 1.1):

1. Клоны кода первого типа – это точная копия исходного кода без каких – либо изменений.

2. Клоны второго типа – это синтаксическая копия исходного кода с внесением изменений в имена или типы переменной, или идентификаторы функции.

3. Клоны кода третьего типа – в скопированном фрагменте кода вносятся изменения в команды, также программист может добавлять или удалять некоторые команды.

Первые два типа клонов можно отыскать с помощью методов лексического или синтаксического анализа. Клоны кода третьего типа могут отличаться своей реализацией, но иметь одинаковый смысл, поэтому для их поиска необходимо применять семантический анализ. Одним из методов, основанных на семантике, является векторная модель семантики (vector space model, VSM). Данная модель используется в информационном поиске для представления коллекции документов векторами из одного общего для всей коллекции векторного пространства. Векторная модель является основой для решения многих задач информационного поиска, таких как: поиск документа

по запросу, классификация документов, кластеризация документов [21]. Применение данного метода для поиска семантически схожих участков кода позволит отыскать клоны кода всех трех типов.

Для создания качественного программного продукта необходимо проводить его анализ на наличие ошибок, в том числе, порождаемых при применении подхода copy-paste programming. Анализируя код на присутствие в нем клонов, можно не только сократить время на исправление ошибок исходного кода во фрагментах – клонах, но и улучшить внутреннюю структуру программы с помощью вынесения схожих по смыслу фрагментов.

Целью данной работы является разработка приложения, основанного на векторной модели семантики и способного отыскать клоны трехадресного кода всех трех типов, возникающих при применении подхода копирования-вставки и модификации исходного кода.

Для достижения данной цели были поставлены и реализованы следующие **задачи**:

1. Провести обзор существующих инструментальных средств, изучить алгоритмы их работы, выявить положительные и отрицательные качества продукта.

2. Изучить и разработать алгоритм, основанный на векторной модели семантики, проверить его эффективность и применимость для поиска клонов в трехадресном коде.

3. Реализовать приложение на основе предложенного алгоритма, проверить правильность и эффективность его работы.

1 КЛОНЫ КОДА

Применение метода копирования и модификации фрагментов исходного кода для уменьшения временных затрат на создание ПО порождает клоны кода.

1.1 Клоны кода и их виды

Клонами программного кода принято считать фрагменты кода, имеющие схожие реализацию, структуру и функционал [4].

Клонами кода первого типа являются точные копии исходного кода без внесения изменений. К таким клонам относятся скопированные фрагменты кода, имеющие синтаксически полное совпадение, за исключением символов пробела, табуляции и комментариев. Пример исходного кода и его клона изображен на рис. 1.

```
//Исходный код
for (int i = 1; i < 5; i++)
{
    f = f * i;
}
//Код - клон
for (int i = 1; i < 5; i++)
{ f = f * i; }
```

Рисунок 1 - Клон кода первого типа

Использование клонов такого рода может применяться программистами для сжатия кода.

Клон кода второго типа – это синтаксически точная копия исходного кода с внесением изменений в имена или типы переменной, или идентификаторы функции. Пример изображен на рис. 2.

```
//Исходный код
for (int i = 1; i < 5; i++)
{
    f = f * i;
}
//Код - клон
for (int k = 0; k < 5; k++)
{
    k = k * v;
}
```

Рисунок 2 - Клон кода второго типа

Данный вид клонов достаточно часто можно встретить при проверке лабораторных работ студентов – копируется код, выполненной ранее работы, вносятся изменения в имена переменных, типы функций, но, к сожалению, смысл написанного остается непонятным для студентов «копипастеров».

Клоны кода третьего типа – это клоны, в которые вносятся какие-либо из изменений: операторы изменяются, добавляются или удаляются. Пример исходного кода и его клона продемонстрированы на рис. 3.

```
//Исходный код
for (int i = 0; i < number; i++)
{
    sum = sum + i;
    //...
}
//Код - клон
for (int i = 0; i < number; i++)
{
    mod = k % i;
    //...
}
```

Рисунок 3 - Клон кода третьего типа

В данном примере в клон исходного кода были внесены изменения, которые приведут к грубейшей ошибке – деление на ноль. В итоге поведение программы становится непредсказуемым.

1.2 Алгоритмы поиска клонов в программном коде

1.2.1 Текстовой подход

Суть данного метода заключается в том, что алгоритм поиска клонов кода считает хеш-коды одной или нескольких строк исходного кода, а затем проводят их сравнение. Если они совпадают, то считается, что сравниваемые строки являются клонами исходного кода. После того, как все клоны найдены, алгоритм может объединять последовательные клоны. Некоторые алгоритмы могут производить сравнение файлов. Для этого рассматривается некое подмножество строк для каждого файла и считают это подмножество его отпечатком. Если два отпечатка совпадают, считается, что соответствующие файлы похожи. Алгоритмы, работающие на основе этого подхода, находят в основном клоны первого типа [6].

1.2.2 Лексический подход

Основанные на данном подходе алгоритмы, в первую очередь получают последовательность лексических единиц – токенов (token) путем разбора исходного кода. Затем производится поиск совпадающей подпоследовательности данных единиц. Для поиска подпоследовательности используют несколько эффективных алгоритмов, которые основаны на параметризованном суффиксом дереве. Алгоритмы, работающие на основе лексического подхода, в основном находят клоны первого и второго типов [6].

1.2.3 Синтаксический подход

Алгоритмы этого типа работают на основе абстрактного синтаксического дерева. Клонами принято считать совпадающие абстрактно синтаксические поддеревья. В некоторых алгоритмах одновременно сравнивается пара деревьев для нахождения совпадающих поддеревьев. Другие же алгоритмы строят суффиксное дерево для каждого поддерева и

сравнивают их. Также существует еще один эффективный алгоритм, который строит векторы для каждого абстрактно синтаксического поддерева и сравнивает эти векторы. Длина вектора зафиксирована и равна количеству всех возможных типов инструкций в абстрактно синтаксическом дереве. Для конкретного поддерева каждый элемент вектора – это количество соответствующих инструкций в этом поддереве. Алгоритмы, работающие на данном подходе, способны отыскать все три типа клонов [6].

1.2.4 Семантический подход

В данном методе используется граф зависимостей программы – это объединенный граф потока данных и графа потока управления, где вершинами являются инструкции программы, а ребрами – зависимости между ними. Существует два основных типа ребер: ребра, выражающие зависимости по данным, и ребра, выражающие зависимости по управлению. Граф зависимостей программы является наиболее общим представлением программы, в нем хранится вся информация о его семантике и структуре, что позволяет более точно анализировать программу. Такой подход используется в задачах оптимизации и поиска семантически схожих участков кода. Как правило, эти методы обладают большой точностью, но весьма медленны [6].

1.2.5 Сравнения подходов

Клоны второго и третьего типов не могут быть найдены лексическим и текстовым подходом. Следовательно, эти подходы не могут быть применены к задачам автоматического рефакторинга. Оставшиеся подходы способны обеспечить поиск клонов всех типов. В общем случае алгоритмы, основанные на графе потока данных, обладают большей точностью, чем алгоритмы, основанные на абстрактном синтаксическом дереве, т.к. в первых хранится информация только о структуре программы, а вторые содержат информацию о семантике и о структуре программы, что, несомненно, дает преимущество. Можно сделать вывод, что высокая точность инструмента поиска клонов кода может быть обеспечена только при использовании

семантического подхода [6].

1.3 Последствия использования клонов кода

Использование при создании нового ПО такого подхода к программированию, как «Copy-paste programming» влечет за собой различного рода проблемы.

На первый взгляд копирование и модификация схожих участков кода, является довольно эффективным методом: сокращаются временные затраты на этапе разработки, что в свою очередь позволяет уделить больше времени на тестирование ПО.

Но применение данного метода разработки чаще всего порождает ряд проблем, на устранение которых могут потребоваться дополнительные временные и финансовые ресурсы.

1.3.1 Ошибки

Возникновение ошибок в программном коде может быть связано с невнимательностью программиста. Желание сделать работу быстрее, усталость откладывают свой отпечаток на качестве программы. При копировании и модификации схожих фрагментов кода довольно часто появляются ошибки в клонах, которые могут быть незаметны на этапе отладки. Поиск таких ошибок обычно происходит на этапе тестирования и может занять довольно большой временной промежуток, на который программист не рассчитывал.

При устранении ошибок в исходном коде, также необходимо вносить изменения в клоны этого фрагмента, на что опять же требуется дополнительное время.

1.3.2 Разрастание кода

Использование клонов программного кода приводит к необоснованному разрастанию программы. Некоторым программистам

проще скопировать и внести небольшие изменения в код, чем вынести функционал в метод, используемый в нескольких местах.

На первый взгляд увеличение количества строк кода не является существенной проблемой. Для небольшой или средней по количеству строк кода программы, наличие клонов чаще всего никак не отражается на времени ее выполнения. Но при работе программиста с кодом клоны могут доставить ряд неудобств: ухудшается вид программы в целом, необходимо фиксировать, где исходный фрагмент кода, а где его клон, иначе можно попросту запутаться и внести изменения не в том месте.

1.3.3 Затруднение поддержки программы

После введения программы в эксплуатацию могут возникать различные ситуации, требующие доработки приложения, например найденные ошибки или расширение функционала.

Вносить изменения в код, имеющий клоны, может быть достаточно тяжело в силу того, что такой код становится менее удобным и читабельным. При изменении функционала можно ошибочно исправить клон, вместо оригинала или наоборот.

1.4 Инструментарии для поиска клонов кода

На данный момент существует довольно большое количество инструментальных средств для поиска клонов в программном коде. Однако алгоритмы, реализованные во многих из них основаны не на семантическом анализе, а на синтаксическом или лексическом. Поэтому большинство приложений способны отыскать только клоны первого и второго типов.

1.4.1 Simian

Приложение Simian [3] разработано для поиска «собезьянненого» кода, т.е. поиска клонов в программном коде. Создатель приложения объясняет необычное название тем, что когда программист копирует чужой код, он ведет себя как обезьяна.

Программа Simian позволяет распознавать клоны в языках C, C++. C#, Java, COBOL, Ruby, JSP, ASP, HTML, XML, Visual Basic, Groovy.

1.4.2 PMD анализатор

PMD [4] анализатор кода предназначен для поиска неиспользуемых переменных, пустых объектов и др. Содержит приложение CPD – Copy Paste Detected, для отыскания дублированного кода.

Дополнение CDP поддерживает проверку кода более чем на десяти языках программирования: Java, C, C++, C#, PHP, Ruby, Fortran, JavaScript, PLSQL, ApacheVelocity, Ruby, Scala, ObjectiveC, Matlab, Python, Go.

1.4.3 CloneDR

CloneDR [5] – данная программа предназначена для поиска клонов всех типов.

Приложение может осуществить поиск клонов кода в языках – C, C++, C#, Java, PHP, HTML, Python.

1.4.4 Visual Studio

Visual Studio Ultimate или Visual Studio Premium [22] помогает найти клонов кода, чтобы можно было выполнить их рефакторинг.

Можно найти клонов конкретного фрагмента или найти все клоны в решении. Помимо обнаружения непосредственных копий, средство анализа клонов может найти фрагменты, которые различаются именами переменных или параметров, а также те, в которых изменилось расположение некоторых операторов.

Анализатор клонов кода находит повторяющийся код в проектах Visual C# и Visual Basic в пределах решения Visual Studio.

1.4.5 Duploc

Инструментальное средство Duploc [11] для поиска клонов кода способно осуществить поиск клонов только первого типа. В качестве клонов данный инструментарий считает полностью идентичные строки.

1.4.6 DuDe

DuDe [18] – программа для поиска всех типов клонов, основанная на тестовом подходе. Клоны третьего типа распознаются с низкой точностью, по сравнению с другими программами [5,6, 7, 9, 2].

1.4.7 Dup

Инструмент Dup [9, **9**, **2**] осуществляет поиск клонов на основе лексического анализа и способен отыскать клоны первого и второго типов. Строится суффиксное дерево из последовательности токенов и находятся максимально совпадающие последовательности.

1.5 Выводы по разделу 1

Преимущества использования клонов кода при написании программ довольно обманчивы на первый взгляд. Программисту кажется, что такой подход как копирование и модификация фрагментов кода может существенно ускорить и облегчить написание программы. На самом деле это не так: приведенные в п. 1.3 последствия использования клонов говорят об обратном.

Для написания качественного ПО программистам следует избегать данного метода написания программ, иначе в процессе работы могут возникнуть дополнительные ошибки, а также проблемы с дальнейшей поддержкой программного продукта [2].

Отыскание клонов в программном коде должно производиться перед этапом тестирования. Это позволит программисту вовремя исправить недочеты в коде и внести соответствующие изменения для улучшения кода программы.

2 ПОИСК КЛОНОВ В ТРЕХАДРЕСНОМ КОДЕ

Используя для анализа трехадресную версию вместо исходного кода можно осуществить поиск клонов всех трех типов. Семантический анализ дает более высокий уровень точности при поиске клонов, по сравнению с синтаксическим и лексическим анализами.

2.1 Трехадресный код

Трехадресный код является промежуточным представлением программного кода. Структура трехадресного кода выглядит следующим образом: в левой части расположены имена переменных, метки перехода или временные объекты, а в правой части может содержаться не более одного оператора. Например, строка кода «result = a*x*x + b*x + c» может быть представлена в виде:

```
t1 = a * x
t2 = t1 * x
t3 = b * x
t4 = t2 + t3
result = t4 + c
```

Трехадресный код является линеаризованным представлением синтаксического дерева. Он построен на основе адресов и команд. В качестве адреса могут быть использованы: имена, константы и сгенерированные компилятором временные переменные. Список наиболее часто встречающихся команд включает в себя: операции присваивания, бинарные арифметические или логические операции, унарные операции, условные и безусловные переходы, вызовы процедур и возврат из них и др.

В компиляторе команды трехадресного кода могут быть представлены как записи с полями для операции и операндов. Такие представления называются четверками, тройками и косвенными тройками [2].

2.1.1 Четверки

Четверка - это запись с четырьмя полями: *op, arg_1, arg_2* и *result*. Поле *op* содержит внутренний код операции. В операторах с унарными операциями типа *x = minus y* или *x = y* поле *arg2* не используется. В некоторых операциях, таких как, например, передача параметра, могут не использоваться ни *arg2*, ни result. Условные и безусловные переходы помещают в *result* целевую метку. На рис. 4, приведены четверки для оператора присваивания a = b *-c + b *-c.

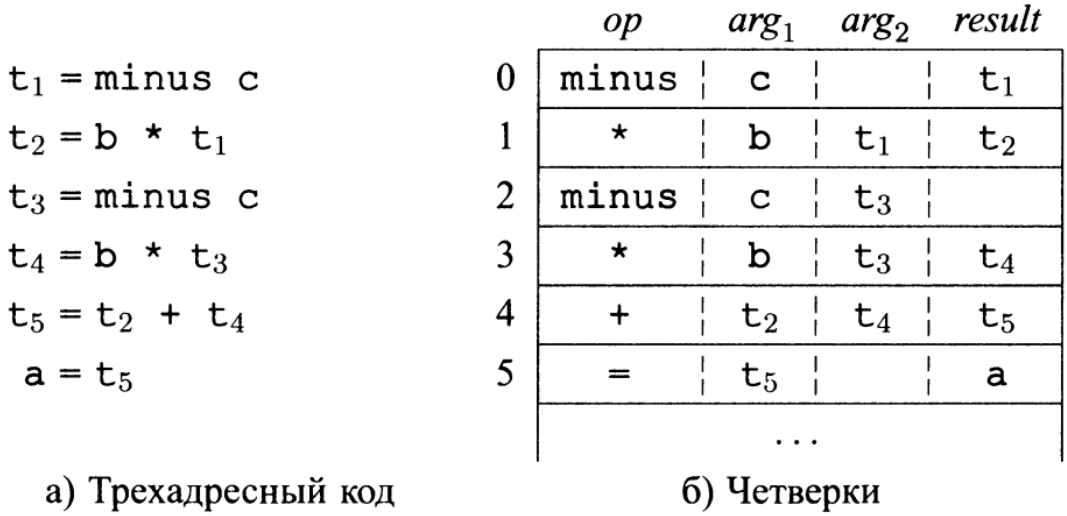

Рисунок 4 – Трехадресный код и его представление с использованием четверок

2.1.2 Тройки

Тройки состоят из трех полей - *op, arg_1* и *arg_2*. Используя тройки, обращение происходит к результату операции *x op y* по ее позиции, а не по явному временному имени. На рис. 4 представление с использованием троек будет обращаться к позиции (0), а не к временной переменной t_1.

Для операций типа *x[i] = y* необходимо две записи в структуре тройки, например *x* и *y* можно поместить в одну тройку, а *y* в другую.

В оптимизирующих компиляторах, где применяется перемещение команд, предпочтительнее использовать четверки, т.к. при перемещении команды, вычисляющей временное значение *t*, не требуется изменение в командах, использующих *t*. В случае троек может потребоваться изменение

всех ссылок на ее результат, потому что обращение к результату операции выполняется с использованием ее позиции.

2.1.3 Косвенные тройки

Структура косвенных троек состоит не из списка самих троек, а из списка указателей на тройки. Например, массив *instruction* предназначен для перечисления указателей на тройки в требуемом порядке. Косвенные тройки будут выглядеть, как показано на рис. 5.

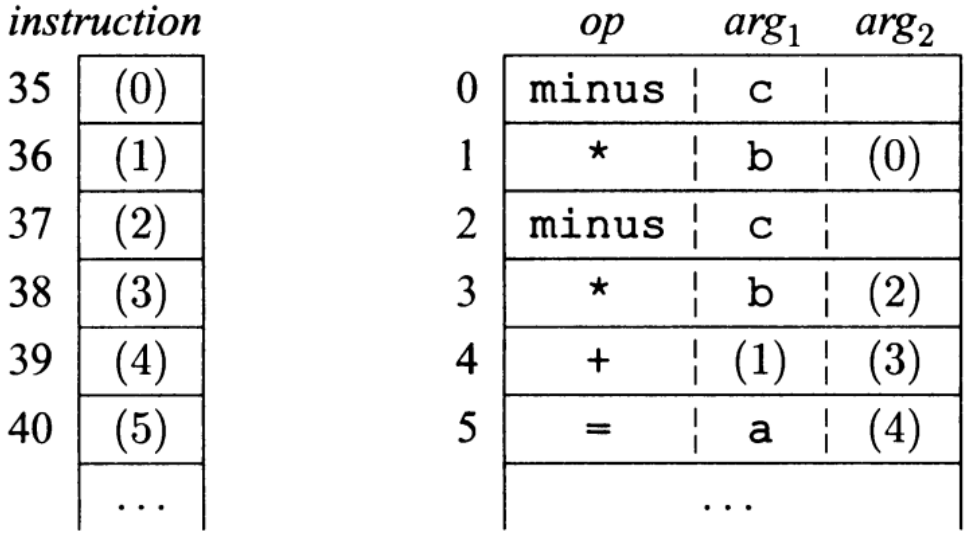

Рисунок 5 – Представление трехадресного кода с применением троек

При использовании косвенных троек оптимизирующий компилятор способен производить перемещение команд с помощью переупорядочивания списка *instruction*, при этом, никак не влияя на сами тройки [2].

2.2 Векторная модель семантики

Основная идея векторной модели семантики – это представление каждого документа коллекции в качестве вектора в векторном пространстве. Чем ближе лежат друг к другу вектора, тем более схожими семантически являются документы. Для поисковых систем это работает по следующей схеме: запрос пользователя рассматривается в качестве вектора, затем документы сортируются в порядке возрастания расстояния до этого вектора и выдаются пользователю.

Векторная модель основывается на гипотезе о статистической семантике (statistical semantics hypothesis): статистические зависимости употребления слов человеком могут быть использованы для нахождения заложенного в них смысла [19].

Основные варианты, по которым строятся модели для поиска сходства: сходство документов, сходство слов и сходство отношений.

2.2.1 Сходство документов: матрица термин-документ

Если поиск необходимо осуществить в большой коллекции документов, представляющей собой большое количество векторов соответственно, наиболее удобно сгруппировать вектора в матрицу таким образом, что каждая строка матрицы представляет собой отдельный термин, а каждый столбец соответствует некоторому документу.

Допустим, в документе содержится некоторое множество терминов $\{a, a, b,$ (порядок в наборе не важен). Тогда ему будет соответствовать вектор x = (2, 1, 3), где первый элемент соответствует числу вхождений в документ термина a, второй – термина b, третий – термина c.

Информационный поиск с использованием матрицы термин-документ (term–document matrix) основывается на следующей гипотезе: оценивание семантического соответствия поискового запроса к полученному документу можно производить путем представления документа и запроса в виде набора

слов (bag of words). Иначе говоря, релевантность запроса зависит от частоты вхождения слов в документ.

Пусть X – является матрицей термин-документ, коллекция состоит из n документов и m уникальных терминов. Тогда матрица X будет состоять m строк и n столбцов. Через w_i обозначим i-ый термин в словаре, через d_j – j-ый документ коллекции. Тогда каждый элемент x_{ij} матрицы представляет собой количество вхождений термина w_i в документ d_j.

Как правило, большинство значений элементов матрицы термин-документ равны 0. Это связано с тем, что документы содержат малую часть терминов из всего словаря.

2.2.2 Сходство слов: матрица слово-контекст

Модель матрицы термин-документ предназначена для нахождения сходства между отдельными документами. Иногда задача поиска сходства слов может заключаться в рамках не всего документа, а каких-либо его частей. Модель семантики, построенная по принципу матрицы слово-контекст (word–context matrix), отражает частоту вхождения терминов в некоторую фразу, предложение, параграф или главу.

Основываясь на гипотезе распределения в лингвистике, слова, встречающиеся в схожих контекстах, стремятся иметь близкий смысл. Поэтому полностью оправданно применение векторной модели семантики для определения схожести слов.

Слово представляется в виде вектора, элементы которого соответствуют числу вхождений в некоторый фрагмент. Семантическое сходство определяется близостью векторов.

2.2.3 Сходство отношений: матрица пара-модель

В матрице пара-модель (pair–pattern matrix) каждая строка должна описывать некоторую пару слов (например, дровосек: топор или швея: ткань), а каждый столбец соответствует модели, которая определяет отношение между парами слов (например, X использует Y).

В данном случае, важную роль играет расширенная гипотеза распределения, согласно которой, отношения, происходящие вместе со схожими парами слов, стремятся иметь близкий смысл. Например, отношение «*X* потерял (что?) *Y*» и «*Y* потеряна (кем?) *X*» имеет склонность употребляться со схожими парами X: Y. В связи с этим, можно предположить, что приведенные отношения имеют схожий смысл.

Основываясь на гипотезе о скрытых связях, пары слов, встречающиеся в похожих моделях, стремятся иметь близкую семантическую зависимость [20]. Степень сходства отношений между словами определяется близостью векторов-строк матрицы пара-модель.

2.2.4 Другие варианты векторных моделей

Основываясь на модели пара-модель, можно составить матрицу тройка-модель, которая устанавливает зависимость между отношениями тройки слов.

Обобщение матричных подходов позволит перейти к составлению тензоров, таких как, термин-документ-язык для многоязычного информационного поиска или глагол-подлежащее-дополнение для изучения предпочтений в сочетаемости слов с глаголами.

2.2.5 Лингвистическая обработка для векторной модели

Предположим, имеется большая коллекция текстов на естественном языке. Прежде, чем сформировать одну из матриц, необходимо провести предобработку текста. Можно выделить три класса предобработки текста. Во-первых, исходный текст следует разметить. Это необходимо, потому что нужно решить, что может являться терминами и как их извлечь из «сырого» текста. Во-вторых, текст нужно нормализовать: привести различные слова к некоторой начальной форме (например, слова деревянный, деревья, деревом необходимо нормализовать в дерево). И, наконец, в-третьих, текст необходимо комментировать. Данная процедура подразумевает создание для

каждого слова метки, указывающей на принадлежность его к определенной части речи.

2.2.5.1 Разметка текста

На первый взгляд разметка текста кажется довольно простой задачей, но на практике это не всегда так.

Программа разметки должна четко знать, как руководствоваться пунктуацией, различать переносы слов и определять составные слова-термины. При этом возможно игнорирование слов с высокой степенью встречаемости в тексте, но с малой информативностью (например, союзы или предлоги).

2.2.5.2 Нормализация текста

Необходимость нормализации текста обусловлена тем фактом, что различные строки могут выражать один и тот же смысл. В связи с этим необходимо приводить слова к единой форме.

Для начала все символы приводятся к одному регистру, при этом необходимо следить за тем, чтобы смысл слова не изменился (например аббревиатура РАН в которой все символы написаны в верхнем регистре, обозначает Российскую Академию Наук, употребление слова в нижнем регистре обозначает родительный падеж от раны).

С морфологической точки зрения, слова могут принимать различные формы (например, изменение глаголов в русском языке по временам и числам и по родам в прошедшем времени). Для отыскания различных форм слова необходимо применять стемминг (стемматизация) – процесс поиска слова во всех его морфологических формах с последующим выделением основы (или корня). Благодаря тому, что в английском языке аффиксы сравнительно просты и постоянны, алгоритмы стемминга, основанные на эвристике, работают довольно хорошо.

В агглютинативном языке (японский, монгольский и др.) словообразование происходит путем добавления к корню различных

формантов, причем каждый из них передает только одно значение. В данном случае осуществить процедуру стемминга гораздо труднее. Одно слово из агглютинативного языка может соответствовать нескольким из русского.

Эффективность работы системы информационного поиска можно измерить двумя показателями: точности и полноты. Показатель точности представляет собой оценку условной вероятности того, что выданный системой документ действительно соответствует запросу. Показатель полноты – это оценка условной вероятности того, что релевантный запросу документ будет выдан системой пользователю.

Нормализация текста способствует увеличению полноты поиска и одновременно снижению его точности. Когда слова приводятся к единой форме, системе проще распознать их сходство, благодаря чему находится большее число подобных документов и полнота возрастает. Однако при упрощении начальный смысл слова может измениться, в результате снизится показатель точность информационного поиска.

Если коллекция документов мала, то достаточно «активная» нормализация текста будет способствовать увеличению полноты. В больших коллекциях часто бывает более важно находить документы с высокой точностью, в связи с чем нормализацию текста проводят в более мягкой форме.

2.2.5.3 Комментирование

Иногда одно и то же слово в тексте может быть употреблено в различных значениях. Общие методы комментирования заключаются в добавлении метод с информацией о части речи, о наличии у слова других значений и синтаксический разбор предложения.

Процедура комментирования, по сути, является обратной операцией к нормализации. Проведение этой процедуры позволяет увеличить точность, однако при этом снижается полнота информационного поиска.

Для языков, где основным способом словообразования является конверсия, комментирование будет довольно актуально. Например, в английском язык из существительного fly (полет) таким образом получается глагол fly (лететь). Написание этих слов совпадает, однако при употреблении в выражениях они могут нести разный смысл. Результат информационного поиска только по глаголу будет иметь более высокую точность, однако в некоторых документах и существительное способно нести в себе требуемый смысл, но система не сможет их найти, вследствие чего снижается полнота поиска.

Для большей информативности для информационного поиска необходимо комментировать запросы с синтаксической и семантической информацией. Синтаксическое комментирование включает сегментацию запросов и разметку частей речи. Это используется для устранения неоднозначности в сокращениях и поиск ассоциаций среди ключевых слов.

Комментирование также применяется для измерения семантической схожести слов и понятий (для моделей, основанных на матрице слово-контекст). В работе [20] представлен алгоритм, который обнаруживает близкий смысл слов путем кластеризации векторов-строк матрицы слово-контекст.

2.2.6 Математическая обработка для векторной модели

После разметки, нормализации и комментировании текста, на первом шаге обработки создается частотная матрица. На втором шаге необходимо настроить веса её элементов, т. к. большинство часто встречающихся слов имеют малую информативность. Затем полученные значения сглаживаются, чтобы ослабить влияния случайного шума и заполнить нулевые элементы в разряженной матрице. И на последнем шаге находятся сходства между векторами.

2.2.6.1 Построение частотной матрицы

Каждый элемент частотной матрицы соответствует некоторому событию: заданное слово (термин, или пара слов) встречается в конкретном документе (контексте, модели) определенное количество раз. Теоретически построение матрицы сводится к простому подсчету количества вхождений слова в текст. На практике могут возникнуть сложности, если коллекция документов большая.

Обычный подход к построению частотной матрицы состоит из двух шагов. На первом шаге последовательно просматриваются все тексты, и записываются события и их частоты в хэш-таблицу или базу данных. На втором шаге, на основании полученных данных строят матрицу частот.

2.2.6.2 Взвешивание элементов

Основная идея для данного шага - чем неожиданнее является событие, тем больший интерес оно представляет. Самый популярный способ взвешивания для модели термин-документ – это tf-idf (term frequency - inverse document frequency), т.е. если в одном документе термин встречается часто, а в других – редко, то такой термин имеет большой вес. В таком случае вес рассчитывается по формуле:

$$w_{ki} = \frac{(1 + \log N_{ik}) * \log\left(\frac{|D|}{N_k}\right)}{\sqrt{\sum_{s \neq k}(\log(N_{is}) + 1)^2}} \qquad (1)$$

где – количество появлений *k*-ого термина в *i*-ом документе, – количество появлений *k*-ого термина во всех документах, – количество документов в коллекции.

Альтернативным способом присваивания весов является метод PMI (Pointwise Mutual Information), который хорошо работает для матриц слово-контекст и матриц термин-*документ*. Positive PMI – это один вариантов PMI,

в котором все отрицательные веса необходимо поменять на ноль. Метод PPMI также подходит для матриц пара-модель.

Пусть F – это матрица частот слово-контекст с – строками и – столбцами. Каждая *i*-ая строка матрицы F – это вектор , соответствующий слову , а *j*-ый столбец матрицы F – это вектор , определяемый контекстом . Применяя метод PPMI, получаем матрицу X – как результат работы с исходной матрицей F. Заметим, что $\dim(X) = \mathrm{di}$. Расчет элементов результирующей матрицы происходит следующим образом:

$$p_{kl} = \frac{f_{kl}}{\sum_{i=1}^{n_r} \sum_{j=1}^{n_c} f_{ij}} \tag{2}$$

$$p_{k^*} = \frac{\sum_{j=1}^{n_c} f_{kj}}{\sum_{i=1}^{n_r} \sum_{j=1}^{n_c} f_{ij}} \tag{3}$$

$$p_{j^*} = \frac{\sum_{j=1}^{n_r} f_{il}}{\sum_{i=1}^{n_r} \sum_{j=1}^{n_c} f_{ij}} \tag{4}$$

$$pmi_{ij} = \log\left(\frac{p_{ij}}{p_{i^*} p_{j^*}}\right) \tag{5}$$

$$x_{ij} = \begin{cases} pmi_{ij}, & pmi_{ij} > 0 \\ 0, & pmi_{ij} \leq 0 \end{cases} \tag{6}$$

В данном случае – это ожидаемая вероятность того, что слово окажется в контексте , – ожидаемая вероятность слова ,

и – вероятность появления контекста . Если и статистически независимы, то $p_{i*}p_{j*}$, и тогда обратится в ноль. С другой стороны, если между конкретным контекстом и словом есть сильная семантическая связь, то следует ожидать, что вероятность будет выше, чем и в этом случае коэффициент примет положительное значение. Это следует из приведенной выше гипотезы распределения. Если некоторое слово совершенно не связано с контекстом , то окажется отрицательным. В методе PPMI представляющие интерес семантические связи отражаются высоким значением элемента , иначе получается нулевое значение.

Недостаток метода PMI заключается в том, что он склонен давать высокие веса нечастым событиям. При этом если некоторое слово встречается только в определенном контексте, то $p_{i*} = p_{j*}$. С уменьшением вероятности появления слова значение возрастает. Одно из решений данной проблемы заключается в введении некоторого штрафующего коэффициента:

$$\theta_{ij} = \frac{f_{ij}}{f_{ij}+1} * \frac{\min\left(\sum_{k=1}^{n_r} f_{kj}, \sum_{k=1}^{n_c} f_{ik}\right)}{\min\left(\sum_{k=1}^{n_r} f_{kj}, \sum_{k=1}^{n_c} f_{ik}\right)+1} \quad (7)$$

Тогда с учетом штрафующего коэффициента, будет равно:

$$newpmi_{ij} = \theta_{ij} * pmi_{ij}. \quad (8)$$

2.2.7 Сглаживание матрицы

Для улучшения выполнения информационного поиска ограничение числа векторных компонент является самым простым способом. Из всех слов выбираются только те слова, которые в наибольшей степени встречаются в контексте. Перед применением этого способа необходимо очистить текст от стоп-слов, которые несут в себе мало полезной информации.

Нахождение сходства между парами всех векторов – это вычислительно сложная задача, поэтому оставляют только наиболее значимые факторы.

Рассмотрим способ, основанный на сингулярном разложении, который позволяет представить X в виде произведения матриц L. Пусть $r = 1$ и (где $k < r$) – диагональная матрица, сформированная из первых k сингулярных чисел, и – матрицы, созданные из соответствующих столбцов U и V. Тогда U_k – матрица ранга k, которая лучшим образом аппроксимирует оригинальную матрицу X. Это означает, что для матрицы $\hat{X} = U_k$ минимизируется выражение $\|\hat{X}$, где – норма Фробениуса.

Данный метод может быть использован для поиска скрытого смысла. Предположим, что X – это матрица слово-контекст. – это линейное отображение между словами и контекстами меньшего ранга, чем X. При понижении ранга матрицы $(k < r)$ остаются только самые «сильные» соответствия.

Также с помощью подобного преобразования можно снизить уровень шума, степени разреженности матрицы X.

2.2.8 Сравнение векторов

2.2.8.1 Косинусное сходство

Косинусное сходство — это мера сходства между двумя векторами предгильбертового пространства, которая используется для измерения косинуса угла между ними.

Если даны два вектора признаков - A и B, то косинусное сходство, $cos(\theta)$, может быть представлено через скалярное произведение и норму:

$$similarity = \cos\theta = \frac{A * B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n}(A_i)^2} \times \sqrt{\sum_{i=1}^{n}(B_i)^2}} \quad (9)$$

{\displaystyle {\text{similarity}}=\cos(\theta)={A\cdot B \over \|A\|\|B\|}={\frac {\sum \limits _{i=1}^{n}{A_{i}\times B_{i}}}{{\sqrt {\sum \limits _{i=1}^{n}{(A_{i})^{2}}}}\times {\sqrt {\sum \limits _{i=1}^{n}{(B_{i})^{2}}}}}}}Для информационного поиска, косинусное сходство двух документов изменяется в диапазоне от 0 до 1, поскольку частота терма (веса tf-idf) не может быть отрицательной. Угол между двумя векторами частоты терма не может быть больше, чем 90°.

Одна из причин популярности косинусного сходства состоит в том, что оно эффективно в качестве оценочной меры, особенно для разреженных векторов, так как необходимо учитывать только ненулевые измерения [19].

2.2.8.2 Мягкая косинусная мера

«Мягкая» косинусная мера [21] — это «мягкая» мера сходства между двумя векторами, то есть мера, которая учитывает сходства между парами признаков. Обычное косинусное сходство рассматривает признаки векторной модели как независимые или полностью обособленные, тогда как «мягкая» косинусная мера учитывает сходства признаков в векторной модели. Это

позволяет обобщить идею косинусной меры, а также идею сходства объектов в векторном пространстве («мягкое» сходство).

Например, в области обработки естественного языка сходство между объектами имеет интуитивный характер. Такие признаки как слова, могут быть довольно схожи, хотя формально они считаются различными признаками в векторной модели. Например, слова «играть» и «игра» различны и, таким образом, отображаются в различных измерениях в векторной модели, хотя, очевидно, что они связаны семантически.

Для расчета «мягкой» косинусной меры вводится матрица *s* сходства между признаками. Она может рассчитываться, используя расстояние Левенштейна или другие меры сходства, например, различные меры сходства в Wordnet. Затем производится умножение с применением данной матрицы.

Если даны два *N*-мерных вектора *a* и *b*, то мягкая косинусная мера рассчитывается следующим образом:

$$soft_{cosine(a,b)} = \frac{\sum_{i,j}^{N} s_{ij} a_i b_j}{\sqrt{\sum_{i,j}^{N} s_{ij} a_i a_j} \sqrt{\sum_{i,j}^{N} s_{ij} b_i b_j}} \qquad (10)$$

где = сходство(признак *i*, признак *j*).

При отсутствии сходства между признаками (= 1, = 0 для $i \neq j$), данное уравнение принимает вид общепринятой формулы косинусного сходства.

Степень сложности этой меры является квадратичной, что делает её вполне применимой к задачам реального мира. При необходимости степень сложности может быть преобразована в линейную.

2.2.8.3 Другие способы сравнения векторов

Один из способов основан на нахождении расстояния между векторами. Тогда схожесть векторов равна:

$$sim(x, y) = \frac{1}{dist(x, y)} \quad (11)$$

Иногда схожесть находят и по такой формуле:

$$sim(x, y) = 1 - dist(x, y) \quad (12)$$

Идея заключает в том, что чем больше расстояние между векторами, тем меньше они похожи друг на друга. Часто в качестве $dist$ используют либо Евклидово расстояние, либо расстояние Манхэттена.

2.3 VSM как метод поиска клонов в трехадресном коде

Анализ трехадресного кода на наличие в нем клонов первого и второго типов осуществить достаточно просто, т.к. трехадресный код изначально является линеаризованным представлением синтаксического дерева [2]. Для отыскания клонов таких типов необходимо просто найти совпадающие поддеревья фрагментов кода [5].

Поиск клонов в трехадресном коде, основанный на векторной модели семантики, позволит находить клоны всех трех типов. Высокая точность нахождения одинаковых по смыслу, но различно реализованных фрагментов достигается за счет того, что статистические зависимости употребления операторов, основных конструкций языка и выражений могут быть использованы для нахождения заложенного в них смысла.

Векторная модель семантики подразумевает три возможных варианта построения моделей для поиска семантического сходства. Рассмотрим возможность построения векторной модели трехадресного кода для каждого из вариантов.

2.3.1 Матрица термин-документ в трехадресном коде

Матрица термин-документ может использоваться при анализе трехадресного кода, если необходимо отыскать клоны в проекте, содержащем большое количество файлов. В данном случае матрица будет строиться таким образом, что каждая строка матрицы представляет собой отдельную команду или операцию некоторого класса, а каждый столбец соответствует классу – файлу проекта. Тогда поиск с использованием матрицы термин-документ будет производиться на основе следующей гипотезы: оценивание семантического соответствия возможного клона к полученному документу производится путем представления класса и фрагмента кода в виде набора команд и операций. Таким образом,

релевантность возможного клона зависит от частоты вхождения его команд и операций в файлы проекта.

Например, имеется три документа d_j (j =), между которыми необходимо произвести поиск клонов:

d_1:

```
t1 := 1 + b
b := h * d
a := 1 + b
c := s + e
```

d_2:

```
res:= 1 - b
k := s % e
b := 1 + k
res := s * e
```

d_3:

```
a := 1 + m
c := a * e
b := 3 + n
r := b * e
s := 4 + b
c := b * e
```

Матрица X состоит из m строк и n столбцов. В данном случае строками матрицы будут выступать не термины, а команды «+», «*», «-» и «%».

Через w_i обозначим i-ю команду в наборе команд, через d_j – j-ый документ коллекции. Тогда каждый элемент x_{ij} матрицы представляет собой

количество вхождений термина w_i в документ d_j. Исходя из этих данных, для приведенного выше примера, получаем матрицу:

$$X = \begin{pmatrix} & d_1 & d_2 & d_3 \\ + & 3 & 1 & 3 \\ * & 1 & 1 & 3 \\ - & 0 & 1 & 0 \\ \% & 0 & 1 & 0 \end{pmatrix}$$

По построенной матрице термин – документ можно сделать следующие выводы: пара документов и имеет большую семантическую схожесть, чем пары и , и соответственно.

Данная модель удобна для поиска клонов среди проектов, в которых каждая функция реализована в виде отдельного файла или класса. Если в одном файле проекта реализовано несколько программных конструкций, то поиск клонов, основанный на матрице термин-документ, не даст желаемых результатов. Это связано в первую очередь с тем, что поиск будет осуществляться между файлами проекта, и не будет производиться деление всего кода на смысловые фрагменты.

2.3.2 Матрица слово-контекст в трехадресном коде

Модель матрицы термин-документ предназначена для нахождения сходства между отдельными файлами одного проекта. Чаще всего, применение задачи поиска клонов кода необходимо для отыскания клонов в рамках не всего документа, а каких-либо его частей. Модель семантики, построенная по принципу матрицы слово-контекст для трехадресного кода, отражает частоту вхождения операций в рамках некоторого фрагмента кода: конструкции (циклы, условия перехода и т.д.), функции или классы. Операция представляется в виде вектора, элементы которой соответствуют числу вхождений в некоторый фрагмент. Семантическое сходство определяется близостью векторов.

Матрица слово-контекст не подразумевает отыскание клонов именно между фрагментами кода. С помощью векторной модели семантики, основанной на данной матрице довольно неудобно находить клоны, т.к. необходимо учитывать семантику смыслового фрагмента кода, а не его отдельной строки.

2.3.3 Матрица пара-модель для трехадресного кода

Матрица пара-модель для трехадресного коды выглядит следующим образом: в данной матрице каждая строка представляет собой пару объектов (переменных), а каждый столбец соответствует операции, производимой над парой объектов (например: $x + y$). С помощью матрицы этого вида можно полностью описать трехадресные инструкции, например код:

t1 := 1 + b

b := h * d

может быть представлен в виде матрицы пара-модель таким образом:

$$X = \begin{pmatrix} & := & + & * \\ t_1, 1 & 1 & 0 & 0 \\ 1, b & 0 & 1 & 0 \\ b, h & 1 & 0 & 0 \\ h, d & 0 & 0 & 1 \end{pmatrix}$$

Для определения сходства между фрагментами кода необходимо вычислить степень близости векторов строк этих фрагментов.

Эта модель хороша для отыскания клонов первого и второго типа – с ее помощью можно довольно точно отыскать синтаксически схожие фрагменты, т.к. матрица строится непосредственно на именах переменных или временных переменных. Клоны кода третьего типа не могут быть найдены с помощью матрицы этого вида, в связи с тем, что в основу построения матрицы заложены имена переменных.

2.3.4 Построение матрицы для трехадресного кода

Вышеперечисленные матрицы способны хорошо отыскивать семантически схожие слова, тексты. Применение данных моделей для поиска

клонов в программном коде не позволяет описать в полной мере программный код с семантической точки зрения. В связи с этим была предложена новая модель матрицы векторной модели семантики.

2.3.4.1 Матрица для отыскания клонов в трехадресном коде

За основу данной матрицы были взяты матрицы слово-контекст и пара-модель. Построение модели для отыскания клонов в трехадресном коде представляется в виде матрицы, строками m которой выступают все левые части трехадресного кода (т.е. имена переменных, метки перехода или временные объекты), а в качестве столбцов n указываются все операторы, находящиеся в правых частях фрагмента трехадресного кода. Элементами матрицы x_{ij} являются 0 либо 1, в зависимости от того, применяется ли данный оператор правой части трехадресного выражения к левой части. Затем производится сравнение векторов.

На первом шаге, для s сравниваемых фрагментов кода выделяются все левые части трехадресного кода и заносятся в соответствующие матрицы, как m_k (k) строк. Далее необходимо вынести все операторы, применяемые в правой части всех фрагментов трехадресного кода. Эти операторы представляют собой набор n столбцов матрицы. Затем происходит заполнение элементов матрицы: если некоторый оператор применяется к строке матрицы, то на их пересечении ставится 1, иначе 0. В результате получаем разреженную матрицу. Это связано с тем что в трехадресном коде в правой части выражения может применяться не более одного оператора.

В качестве примера рассмотрим трехадресные версии кода для нахождения результата квадратичного и кубических уравнений.

Матрица строки кода квадратичного уравнения «square = a*x*x + b*x + c», представленного в трехадресном виде:

```
t1 = a * x
t2 = t1 * x
t3 = b * x
```

t4 = t2 + t3

square = t4 + c

выглядит следующим образом:

$$X_1 = \begin{pmatrix} & * & + \\ t1 & 1 & 0 \\ t2 & 1 & 0 \\ t3 & 1 & 0 \\ t4 & 0 & 1 \\ \text{square} & 0 & 1 \end{pmatrix}$$

Матрица строки кода кубического уравнения «cubic = a*x*x*x + b*x*x + c*x+d», представленного в трехадресном виде:

t1 = a * x

t2 = t1 * x

t3 = t2 * x

t4 = b * x

t5 = t4 * x

t6 = c * x

t7 = t3 + t5

t8 = t7 + t6

cubic = t8 + d

выглядит следующим образом:

$$X_2 = \begin{pmatrix} & * & + \\ t1 & 1 & 0 \\ t2 & 1 & 0 \\ t3 & 1 & 0 \\ t4 & 1 & 0 \\ t5 & 1 & 0 \\ t6 & 1 & 0 \\ t7 & 0 & 1 \\ t8 & 0 & 1 \\ \text{cubic} & 0 & 1 \end{pmatrix}$$

Сравнение векторов производится по строкам фрагментов с помощью косинусного сходства. Например, косинус угла, между каждой пары строк-векторов матрицы X_1: t1, t2, t3 и строк-векторов матрицы X_2: t1, t2, t3, t4, t5, t6

будет равен единице. Сравнение производится независимо от того, какое имя переменной указано в строке матрицы. За счет этого поиск клонов, основанный на данной матрице, позволяет находить клоны третьего типа, т.к. не учитываются синтаксические имена переменных.

2.3.4.2 Сравнение фрагментов для отыскания клонов в программном коде

После построения матриц для сравниваемых фрагментов (см. п. 2.3.4.1) необходимо провести анализ полученных данных. Если два фрагмента кода имеют заданную точность, то они считаются клонами.

Сравнение фрагментов происходит следующим образом: для вектора i – ой ($i =$) строки матрицы X_1, вычисляется сходство с j – ой ($j =$) строкой матрицы X_2. Сходство векторов определяется с помощью косинусной меры. Если косинус угла между векторами равен 1, то данные строки считаются идентичными. Для определения схожести фрагментов вычисляется количество подряд идущих совпадающих строк.

Косинусное сходство для векторов вычисляется по формуле:

$$similarity = \cos\theta = \frac{A * B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n}(A_i)^2} \times \sqrt{\sum_{i=1}^{n}(B_i)^2}} \qquad (13)$$

{\displaystyle {\text{similarity}}=\cos(\theta)={A\cdot B \over \|A\|\|B\|}={\frac {\sum \limits _{i=1}^{n}{A_{i}\times B_{i}}}{{\sqrt {\sum \limits _{i=1}^{n}{(A_{i})^{2}}}}\times {\sqrt {\sum \limits _{i=1}^{n}{(B_{i})^{2}}}}}}}В данном случае косинусное сходство двух векторов не изменяется в диапазоне от 0 до 1, а может только принимать значения 0 или 1. Это связано с тем, что в правой части трехадресного кода может применяться только один оператор. В связи с этим, все элементы

строки-вектора примут нулевые значения, кроме элемента, находящегося на пересечении со столбцом, оператор которого применяется для данной строки.

Применение косинусного сходства довольно эффективно в качестве оценочной меры, т.к. в данном методе используются разреженные вектора и необходимо учитывать только ненулевые измерения [19].

Вычислим схожесть фрагментов, представленных в п. 2.3.4.1 в качестве примера. Имеется две матрицы и , между которыми необходимо определить сходство:

$$X_1 = \begin{pmatrix} & * & + \\ t1 & 1 & 0 \\ t2 & 1 & 0 \\ t3 & 1 & 0 \\ t4 & 0 & 1 \\ \text{square} & 0 & 1 \end{pmatrix}; X_2 = \begin{pmatrix} & * & + \\ t1 & 1 & 0 \\ t2 & 1 & 0 \\ t3 & 1 & 0 \\ t4 & 1 & 0 \\ t5 & 1 & 0 \\ t6 & 1 & 0 \\ t7 & 0 & 1 \\ t8 & 0 & 1 \\ \text{cubic} & 0 & 1 \end{pmatrix}$$

1. Рассчитаем косинусное сходство между i – ым ( ) вектором матрицы X_1 и j – ым ( ) вектором матрицы X_2. Для этого сперва найдем скалярное произведение векторов:

$$\overline{x_1^1} \cdot \overline{x_1^2} = x_{1x}^1 \cdot x_{1x}^2 + x_{1y}^1 \cdot x_{1y}^2 = 1 \cdot 1 + 0 \cdot 0 = 1$$

где и - координаты первого вектора матрицы X_1, а и координаты первого вектора матрицы X_2.

Затем вычислим длины векторов:

$$\left|\overline{x_1^1}\right| = \sqrt{(x_{1x}^1)^2 + \left(x_{1y}^1\right)^2} = \sqrt{1+0} = 1,$$

$$\left|\overline{x_1^2}\right| = \sqrt{(x_{1x}^2)^2 + \left(x_{1y}^2\right)^2} = \sqrt{1+0} = 1.$$

Вычисляем косинус угла между векторами:

$$\cos\theta = \frac{\overline{x_1^1} \cdot \overline{x_1^2}}{|\overline{x_1^1}| \times |\overline{x_1^2}|} = \frac{1}{1 \times 1} = 1.$$

Значение косинуса равно 1, следовательно, вектора идентичны.

2. Т.к. вектора на предыдущем шаге схожи, то рассчитываем косинусное сходство между i – ым ( ) вектором матрицы X_1 и j – ым ( ) вектором матрицы X_2. Найдем скалярное произведение векторов:

$$\overline{x_2^1} \cdot \overline{x_2^2} = x_{2x}^1 \cdot x_{2x}^2 + x_{2y}^1 \cdot x_{2y}^2 = 1 \cdot 1 + 0 \cdot 0 = 1$$

где и - координаты второго вектора матрицы X_1, а и координаты второго вектора матрицы X_2.

Затем вычислим длины векторов:

$$|\overline{x_2^1}| = \sqrt{(x_{2x}^1)^2 + (x_{2y}^1)^2} = \sqrt{1+0} = 1,$$

$$|\overline{x_2^2}| = \sqrt{(x_{2x}^2)^2 + (x_{2y}^2)^2} = \sqrt{1+0} = 1.$$

Вычисляем косинус угла между векторами:

$$\cos\theta = \frac{\overline{x_2^1} \cdot \overline{x_2^2}}{|\overline{x_2^1}| \times |\overline{x_2^2}|} = \frac{1}{1 \times 1} = 1.$$

Получаем значение косинуса равное 1, следовательно, вектора идентичны.

3. На шаге 2 мы вектора оказались схожи, значит, продолжаем рассчитывать косинусное сходство между i – ым ( ) вектором матрицы X_1 и j – ым ( ) вектором матрицы X_2. Находим скалярное произведение векторов:

$$\overline{x_3^1} \cdot \overline{x_3^2} = x_{3x}^1 \cdot x_{3x}^2 + x_{3y}^1 \cdot x_{3y}^2 = 1 \cdot 1 + 0 \cdot 0 = 1$$

где и - координаты третьего вектора матрицы X_1, а и координаты третьего вектора матрицы X_2.

Вычисляем длины этих векторов:

$$\left|\overline{x_3^1}\right| = \sqrt{(x_{3x}^1)^2 + \left(x_{3y}^1\right)^2} = \sqrt{1+0} = 1,$$

$$\left|\overline{x_3^2}\right| = \sqrt{(x_{3x}^2)^2 + \left(x_{3y}^2\right)^2} = \sqrt{1+0} = 1.$$

Вычисляем косинус угла между векторами:

$$\cos\theta = \frac{\overline{x_3^1} \cdot \overline{x_3^2}}{\left|\overline{x_3^1}\right| \times \left|\overline{x_3^2}\right|} = \frac{1}{1 \times 1} = 1.$$

Получаем значение косинуса равное 1, следовательно, вектора идентичны.

4. На шаге 3 вектора также оказались схожи, продолжаем рассчитывать косинусное сходство между i – ым ( ) вектором матрицы X_1 и j – ым ( ) вектором матрицы X_2. Находим скалярное произведение векторов:

$$\overline{x_4^1} \cdot \overline{x_4^2} = x_{4x}^1 \cdot x_{4x}^2 + x_{4y}^1 \cdot x_{4y}^2 = 0 \cdot 1 + 1 \cdot 0 = 0$$

где и - координаты третьего вектора матрицы X_1, а и координаты третьего вектора матрицы X_2.

Числитель для формулы вычисления косинуса угла между векторами равен 0, следовательно, и значение косинуса тоже, значит, вектора различны. Первый клон фрагментов кода найден.

5. Далее проводим сравнение между i – ым ( ) вектором матрицы X_1 и j – ым ( ) вектором матрицы X_2.

Вектора-строки матрицы X_1 (i) и вектора-строки матрицы X_2 (j) также имеют меру сходства, равную 1. Второй клон между фрагментами кода найден.

6. Дальнейший поиск клонов осуществляется аналогичным образом.

В данном примере не проводилось разбиение на смысловые фрагменты, в связи с этим в самом клоне содержатся идентичные строки.

2.4 Выводы по разделу 2

В работе [3] рассматривалась возможность применения векторной модели семантики для отыскания клонов кода. На основе этого в данном разделе была детально рассмотрена векторная модель семантики, а также возможность применения ее к поиску клонов в трехадресном коде. Основные модели построения векторной модели семантики оказались недостаточно подходящими для отыскания клонов всех типов. В связи с этим, на основе трех существующих моделей была предложена новая, способная найти не только синтаксически и лексически точные, но и семантически схожие фрагменты программного кода.

3 ПРЕДЛОЖЕННЫЙ МЕТОД ПОИСКА КЛОНОВ В ТРЕХАДРЕСНОМ КОДЕ

Реализация метода поиска клонов в трехадресном коде, основанного на векторной модели семантики, подразумевает собой несколько этапов. На первом этапе необходимо подготовить исходный трехадресный код для анализа – произвести его разбиение на фрагменты. На следующем этапе отсеиваются заведомо несхожие фрагменты. Далее непосредственно производится сравнение фрагментов, которые возможно имеют схожую семантику.

3.1 Разделение кода на фрагменты

Для того чтобы провести анализ кода на наличие в нем клонов, необходимо исходный код разделить на фрагменты. Данные фрагменты должны представлять собой смысловую единицу кода – семантически самостоятельный фрагмент.

Задача поиска смысловых единиц кода играет довольно весомую часть для дальнейшего поиска клонов. От того насколько точно разделен исходный код на семантические фрагменты будет зависеть точность поиска клонов среди этих фрагментов. Если семантическая единица будет содержать только часть клона, то он найдется частично или вообще не будет найден.

Разделение кода на смысловые единицы, возможно реализовать несколькими методами. Один из них разделяет граф на слабосвязанные подграфы [23], которые и являются семантической единицей. Однако этот метод имеет недостаток – размеры фрагментов могут сильно отличаться. Например, фрагмент может быть настолько большим, что он будет содержать в себе фрагменты кода, которые могут быть потенциальными клонами. Такой пример рассматривался при вычислении косинуса угла в п. 2.3.4.2. Другой

метод разделяет граф на подграфы таким образом, что любые два из подграфов могут иметь не более *N* общих ребер [17].

Для отыскания смысловых единиц в данной работе были реализованы оба метода. Изначально задается максимально возможное число *N* общих ребер для подграфов. Затем производится разделение исходного графа на подграфы, с учетом показателя *N*.

Применяя оба метода для отыскания смысловых единиц, можно добиться более точного разделения исходного кода на семантические фрагменты, чем применение этих методов по отдельности.

3.2 Поиск подходящих для анализа фрагментов

Чаще всего большинство фрагментов исходного кода не являются клонами. В данном случае нет необходимости сравнивать между собой все фрагменты – на этапе отбора достаточно выбрать схожие по определенным признакам пары, например по количеству попарно совпадающих операторов, применяемых в правых частях трехадресного кода.

Для поиска подходящих для анализа фрагментов удобно использовать одну из моделей векторной семантики – матрицу термин-документ. В данном случае матрица будет строиться таким образом, что коллекция строк матрицы будет состоять из множества смысловых единиц (см. п. 3.1), а в качестве столбцов будет выступать набор всех команд или операций, содержащихся в этих смысловых единицах. Схожесть смысловых единиц определяется частотой вхождения команд и операций одного фрагмента в другой.

Например, имеется три фрагмента, в каждом из которых содержатся следующие наборы команд:

$$f_1 = \{*, +, +, +, *, -\}$$

$$f_2 = \{+, +, *, *, -\}$$

$$f_3 = \{-, +, -, +, -\}$$

Тогда матрица будет состоять из трех строк, четырех столбцов и примет вид:

$$F = \begin{pmatrix} & * & + & - \\ f_1 & 2 & 3 & 1 \\ f_2 & 2 & 2 & 1 \\ f_3 & 0 & 2 & 3 \end{pmatrix}$$

Сравнивая попарно каждую из строк, с помощью косинусного сходства (см. п. 2.3.4.2), получим следующие результаты:

$$\cos\left(\widehat{\overrightarrow{f_1}, \overrightarrow{f_2}}\right) = 0{,}9799;$$

$$\cos\left(\widehat{\overrightarrow{f_1}, \overrightarrow{f_3}}\right) = 0{,}6671;$$

$$\cos\left(\widehat{\overrightarrow{f_2}, \overrightarrow{f_3}}\right) = 0{,}6471.$$

Таким образом, наибольшим сходством обладают фрагменты и . Для них целесообразно проводить дальнейший поиск клонов. Осуществлять сравнение между парами фрагментов (,) и (,) не имеет смысла, т.к. вероятность того, что они могут являться клонами, не на много превышает половину.

3.3 Построение векторной модели семантики

Векторная модель семантики для данного метода поиска клонов кода основывается на отыскании семантически схожих фрагментов кода. В качестве начальных данных для построения модели используются уже предобработанные данные: исходный трехадресный код делится на фрагменты (смысловые единицы), затем производится отсев заведомо несхожих фрагментов. Когда становятся известны возможные клоны, можно приступать к более детальному их рассмотрению – построению векторной модели семантики.

После изучения существующих векторных моделей семантики, как уже говорилось в п. 2.3.4, для трехадресного кода необходима своя модель. Это обусловлено в первую очередь спецификой трехадресного кода. Исходный код обычной программы может рассматриваться как обычный текст, каждый фрагмент которого обладает своей семантикой. Трехадресный код представляет собой структуру, в левой части которой расположены имена переменных, метки перехода или временные объекты, а в правой части может содержаться не более одного оператора. Применение стандартных моделей VSM в данном случае не даст желаемого результата: при построении одной из моделей отыскиваются не все типы клонов, другая модель недостаточно полно описывает семантику кода, третья модель может содержать клоны в возможных клонах и при этом не находить их.

Векторная модель семантики для трехадресного кода была построена на основе матриц слово-контекст и пара-модель. Матрица для модели поиска клонов в трехадресном коде содержит в себе информацию об именах переменных, метках перехода или временных объектах, а также операторах, применяемых к этим объектам. Этих данных достаточно для определения клонов всех трех типов.

Матрица векторной модели семантики трехадресного кода является разреженной матрицей, т.к. в трехадресном коде в правой части выражения может применяться не более одного оператора. Эффективным методом оценивания разреженных векторов является косинусное сходство. Это связано с тем, что при его применении учитываются только ненулевые измерения, а вектора построенной матрицы могут содержать лишь один ненулевой элемент.

На заключительном этапе попарно проводится сравнение векторов с помощью косинусного сходства и на основе полученных результатов делаются выводы о семантической зависимости фрагментов, т.е. о наличии клонов в коде.

3.4 Сравнение фрагментов

Сравнение фрагментов кода основывается на косинусном сходстве. Фрагменты кода, которые являются потенциальными клонами, подлежат более детальному сравнению. Для каждого фрагмента строится матрица векторной модели семантики, затем производится сравнение между строками фрагментов. При измерении сходства векторов-строк применяется косинусная мера – вычисление косинуса угла между этими векторами. Общая структура вычисления косинусной меры для векторов-строк матрицы векторной модели семантики описана в п. 2.3.4.2.

3.5 Выводы по разделу 3

Опираясь на специфику построения трехадресного кода, в данном разделе был предложен метод поиска клонов кода, основанный на векторной модели семантики. Существующие матрицы VSM не обладают достаточными качествами для построения матрицы для трехадресного кода. В связи с этим была построена новая матрица, удовлетворяющая параметрам, необходимым для отыскания клонов всех трех типов.

Для усовершенствования метода поиска клонов трехадресного кода была добавлена двухэтапная предобработка начальных данных, включающая в себя разделение исходного кода на смысловые единицы (семантические фрагменты) и выделение возможных клонов из всех фрагментов. Это позволит сократить временные затраты на поиск приблизительно на четверть, т.к. по некоторым данным [24, 13] число строк клонов кода от общего числа строк, составляет не более 20%.

4 ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПОИСКА КЛОНОВ В ТРЕХАДРЕСНОМ КОДЕ

Разработанное приложение для поиска клонов в трехадресном коде реализовано посредством инструмента Microsoft Visual C#, имеет графический интерфейс.

4.1 Алгоритмы программы

4.1.1 Общая схема построения программы

Общая схема отражает построение основных этапов процесса отыскания клонов в трехадресном коде (рис. 4.1). Программа состоит из следующих разделов: разделение кода на смысловые фрагменты (см. п. 3.1), отбор фрагментов – возможных клонов (см. п. 3.2), построение векторной модели семантики для отобранных фрагментов (см. п. 2.3.4.1) и сравнение фрагментов для выявления наличия клонов (см. п. 2.3.4.2).

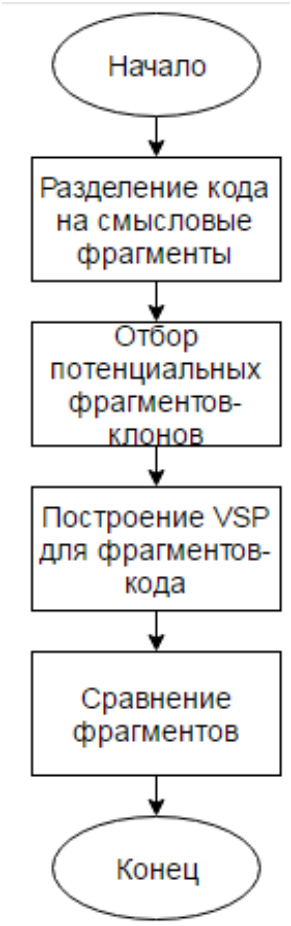

Рисунок 6 – Общая схема программы

4.1.2 Алгоритм деления исходного кода на фрагменты

Алгоритм деления исходного трехадресного кода на семантические единицы описан в п. 3.1 данной работы. На рис. 7 представлена схематическая реализация алгоритма.

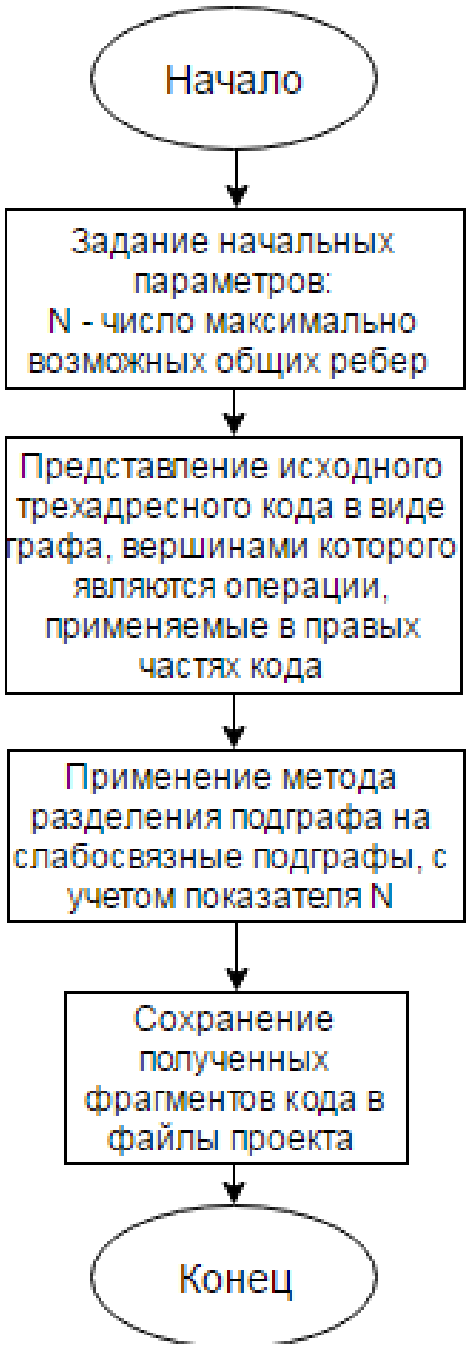

Рисунок 7 – Алгоритм деления исходного кода на смысловые единицы

4.1.3 Алгоритм отыскания из фрагментов потенциальных клонов

Имеется *k* исходных фрагментов, полученных на этапе деления кода. Пусть *mass_F[k]* – массив исходных фрагментов, тогда схема алгоритма отыскания из фрагментов потенциальных клонов будет выглядеть, как на рис. 8. Алгоритм применения VSM для данной схемы представлен в разделе 4.1.4.

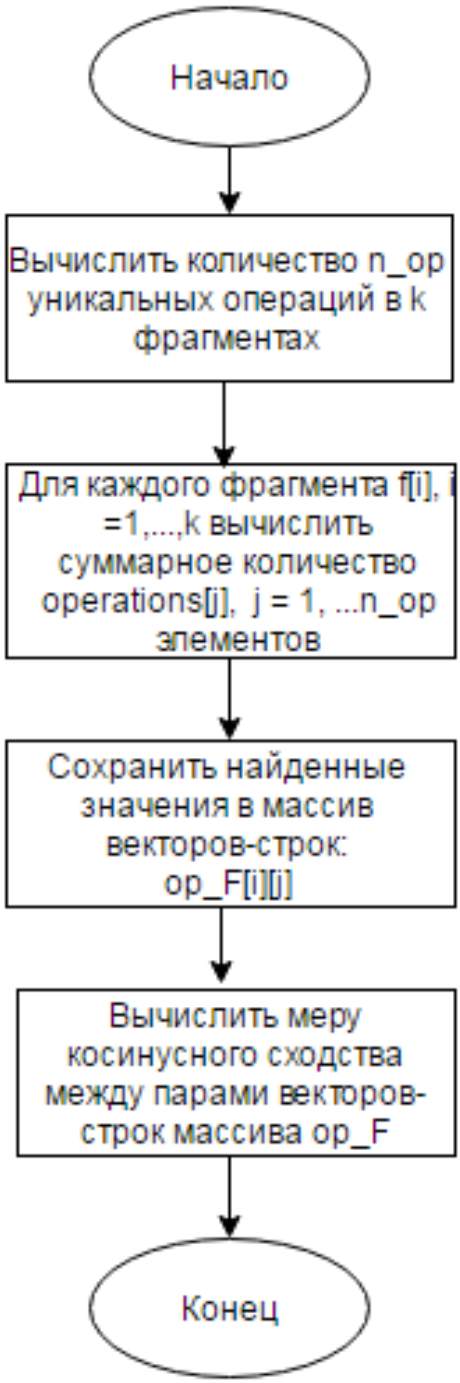

Рисунок 8 – Алгоритм отыскания из фрагментов потенциальных клонов

4.1.4 Алгоритм поиска потенциальных клонов среди отобранных фрагментов

Блок-схема поиска потенциальных клонов среди отобранных фрагментов изображена на рис. 9. Изначально задается параметр сравнения *e* – минимально допустимая мера для определения сходства фрагментов.

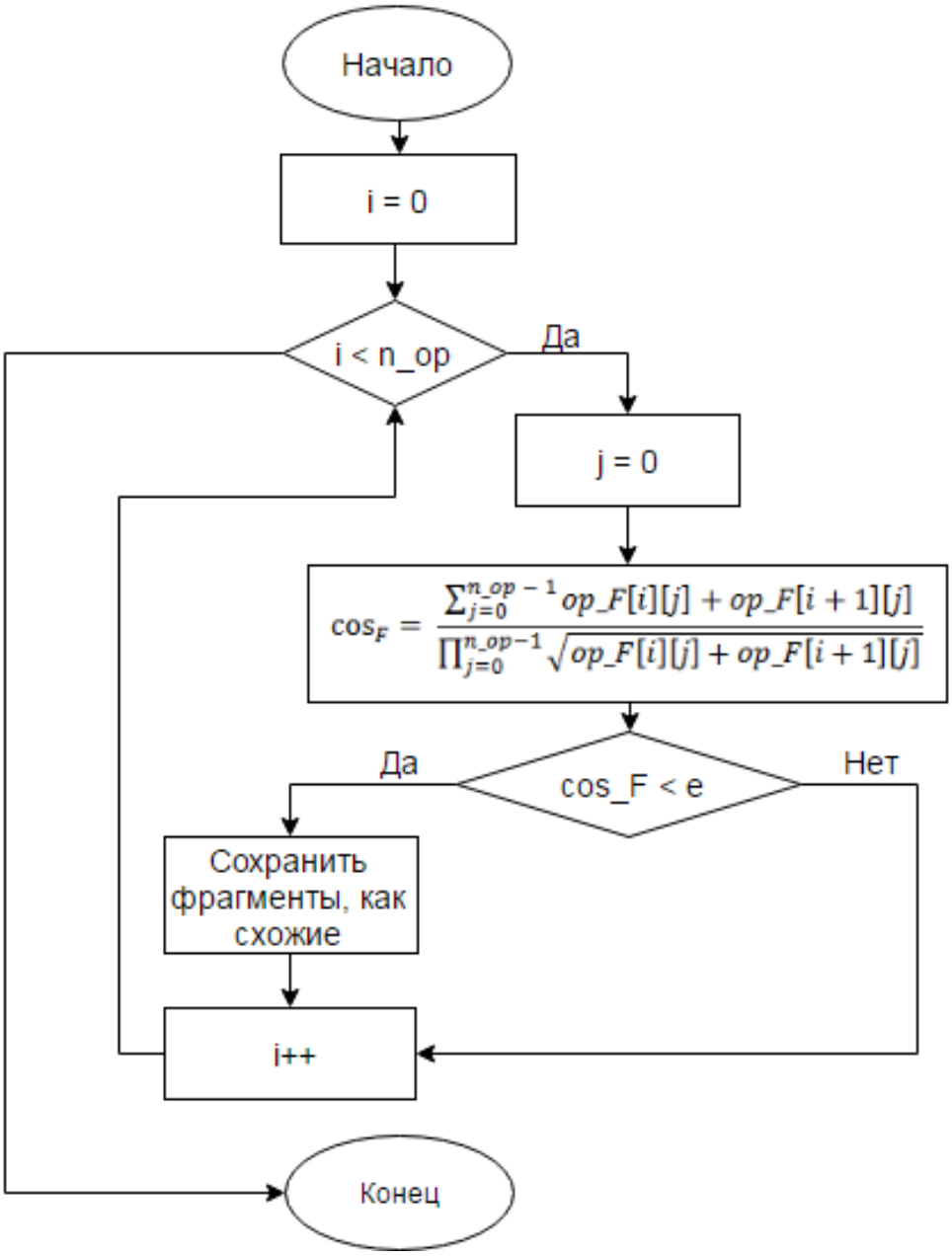

Рисунок 9 – Алгоритм применения VSM для нахождения потенциальных клонов

4.1.4 Алгоритм применения VSM для поиска клонов кода

Алгоритм построения VSM для потенциальных клонов и алгоритм поиска клонов среди отобранных фрагментов изображен на рис. 10.

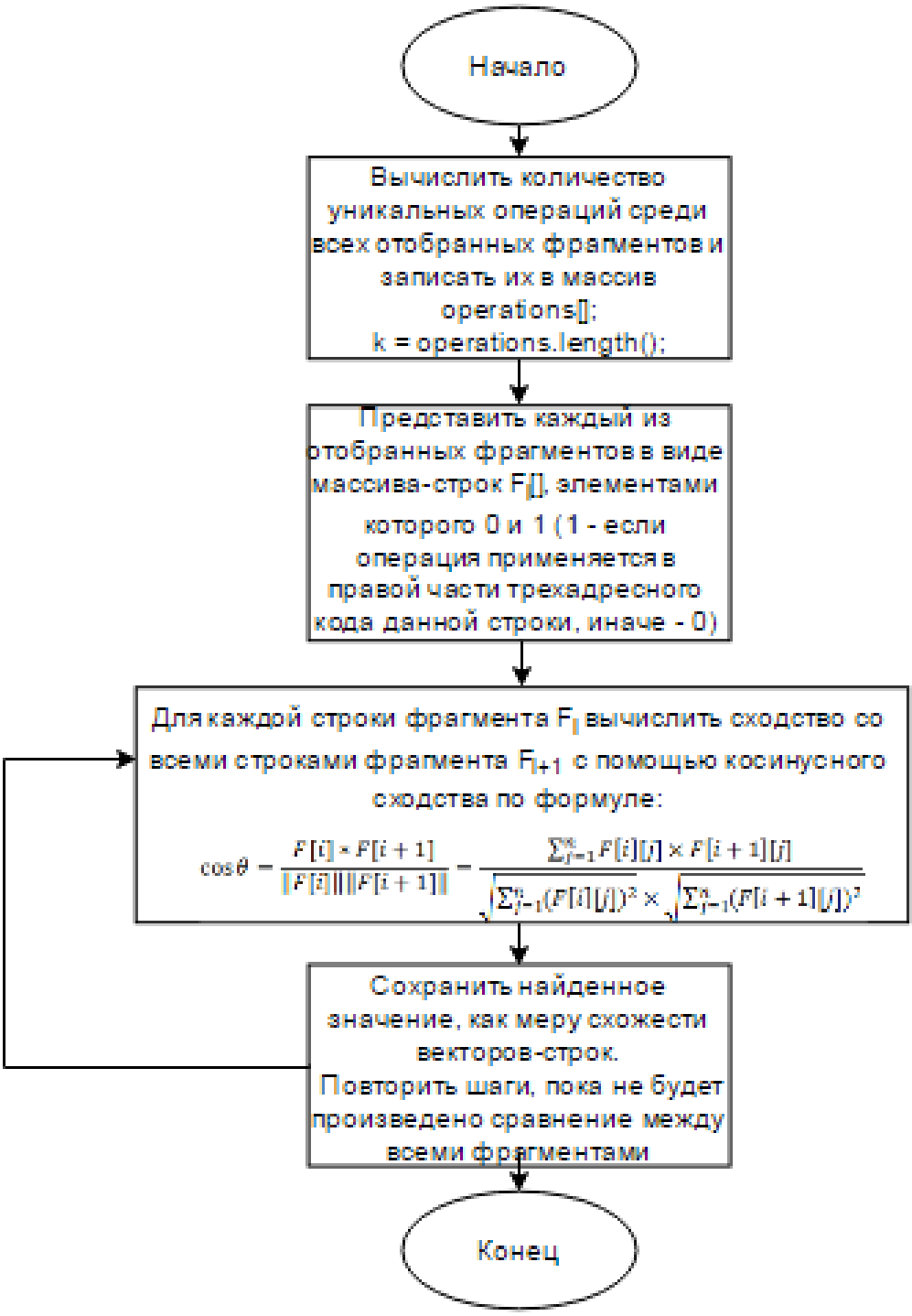

Рисунок 10 – Алгоритмы построения VSM и поиска клонов

4.2 Код программы

4.2.1 Реализация деления исходного кода на фрагменты

Алгоритм деления исходного кода на фрагменты

ВХОД:
N – число максимально возможных общих ребер между подграфами;
code_str[] – массив строк исходного трехадресного кода
ВЫХОД:
массив слабосвязных подграфов

```
for (int i = 0; i < code_str.length(); i++)
    построить ветку графа G(V, E), вершиной s □ V которого является
    операция, применяемая в правой части соответствующей строки
    трехадресного кода
    i++
end for

//Применение метода разделения графа на слабосвязные подграфы
GraphPartition(G(V, E))
```

4.2.2 Реализация поиска потенциальных клонов

Алгоритм поиска потенциальных клонов

ВХОД:
op_F[i][] – массив операций фрагмента F[i];
e – точность поиска потенциальных клонов
ВЫХОД:
Набор фрагментов кода – потенциальных клонов

for (int i = 0; i < F[i].length(); i ++)
for (int j = 0; i < F[i].length(); j ++)

$$\cos_F = \frac{\sum_{j=0}^{n\_op-1} op\_F[i][j] + op\_F[i+1][j]}{\prod_{j=0}^{n\_op-1} \sqrt{op\_F[i][j] + op\_F[i+1][j]}}$$

$\cos_F$ > e
сохранить как схожие фрагменты
else
end for
end for

4.2.3 Реализация поиска клонов

Алгоритм поиска клонов среди потенциальных клонов

ВХОД:
F[i] – набор фрагментов потенциальных клонов (i = 1, …, m);
 – минимальная мера схожести клонов;
operations[] – массив уникальных операций всех фрагментов F[i]
ВЫХОД:
Набор клонов трехадресного кода

```
for (int i = 0; i < m; i ++)
   //Применить метод построения VSM для F[i]
   VSM(F[i])
end for

for (int i = 0; i < m - 1; i ++)
   for (int j = 0; i < F[i].length(); j ++)
      //Вычислить меру косинусного сходства между векторами, по
      формуле:
```

$$\cos\theta = \frac{F[i] * F[i+1]}{\|F[i]\|\|F[i+1]\|} = \frac{\sum_{j=1}^{n} F[i][j] \times F[i+1][j]}{\sqrt{\sum_{j=1}^{n}(F[i][j])^2} \times \sqrt{\sum_{j=1}^{n}(F[i+1][j])^2}}$$

cos

```
      Клоны найдены, сохранить пару фрагментов
      else
   end for
end for
```

ЗАКЛЮЧЕНИЕ

В ходе проведения научного исследования была выявлена возможность построения анализатора кода, позволяющего находить клоны кода в трехадресном коде. В работе были рассмотрены типы клонов, алгоритмы их поиска, предложено и реализовано решение для отыскания клонов в трехадресном коде.

Применение векторной модели семантики для поиска схожих фрагментов подразумевает построение матрицы, на основе которой высчитывается мера схожести фрагментов. Стандартные матрицы VSM не позволяют достаточно полно описать структуру трехадресного кода для дальнейшего анализа. В связи с этим была разработана новая модель матрицы, в которой учитывается специфика трехадресного кода.

Также на основе VSM было предложено решение для предобработки трехадресного кода – разделение исходного графа программы на слабосвязные подграфы. В данном случае векторная модель семантики применялась для отыскания потенциальных клонов, что позволяет более точно найти клоны, затратив на поиск меньше времени.

По завершении достижения поставленных задач, была реализована цель магистерской работы - разработано приложение, основанное на векторной модели семантики, которое способного отыскать клоны трехадресного кода всех трех типов, возникающих при применении подхода копирования-вставки и модификации исходного кода.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. **Аллен, Э.** Типичные ошибки проектирования [Текст] / Эрик Аллен; перевод А. Леонтьев. — Издательский дом «Питер», 2003. — 224 с. — ISBN 5-887827-304-6.

2. **Ахо, А.** Компиляторы. Принципы, технологии и инструментарий [Текст] / Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман; перевод Игорь Красиков. — Вильямс, 2016. — 1184 с. — ISBN 978-5-8459-1932-8.

3. **Бабкина, А.А.** Клоны кода, как следствие использования методологического антипаттерна «Copy-paste-programming» [Текст] / А.А. Бабкина, А.С. Кузнецов, В.В. Кукарцев, г. Красноярск // Компьютерные технологии и телекоммуникации – 2016: сб. тр. / Мин. обр. и науки РФ, Грозненский гос. нефтяной технич. ун. им. акад. М.Д. Миллионщикова, Фак. автоматизации и пр. информ. – Грозный, 2016. – С. 67–70.

4. **Зельцер, Н.Г.** Поиск повторяющихся фрагментов исходного кода при автоматическом рефакторинге [Текст] / Н.Г. Зельцер // Труды Института системного программирования РАН. - 2013. - №25. - С. 39 – 50.

5. **Кристофидес, Н** Теория графов. Алгоритмический подход [Текст] / Н. Кристофидес. – Мир, 1978. – 432 с.

6. **Саргсян, С.** Масштабируемый инструмент поиска клонов кода на основе семантического анализа программ [ТЕКСТ] /С. Саргсян, Ш. Курмангалеев, А. Белеванцев, А. Асланян, А. Балоян// Труды Института системного программирования РАН.- 2015. - Т. 27. - № 1. - С. 39-50.

7. **Baker, B.** A program for identifying duplicated code [Текст] / B. Baker // Computing Science and Statistics: 24th Symposium on the Interface. - 1992. - Т. 24. - С. 49-57.

8. **Baker, B.** On finding duplication and near-duplication in large software systems [Текст] / B. Baker // 2nd Working Conference on Reverse Engineering (WCRE). - 1995. - С. 86-95.

9. **Baker, B.** Parameterized pattern matching: Algorithms and applications [Текст] / B. Baker // Journal Computer System Science. - 1996. - Т. 52. - № 1. - С. 28-42.

10. **Christopher, D.** An Introduction to Information Retrieval [Текст] /Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze Draft // Cambridge University Press. - 2009. – С. 544 .

11. **Jurafsky, D.** Speech and Language Processing. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition [Текст] / Daniel Jurafsky, James H. Martin // Second Edition. Pearson Education International. – 2009. – С. 1024 pp.

12. **Ducasse, S.** A language independent approach for detecting duplicated code [Текст] / S. Ducasse, M. Rieger, S. Demeyer. // 15th International Conference on Software Maintenance (ICSM). – 1999. – С. 109-119.

13. **Li, Z.O.** A metric space based software clone detection approach [Текст] / Z.O. Li, J. Sun // 2nd International Conference on Software Engineering and Data Mining. – 2010. – С. 111-116.

14. **Maeda, K.** Syntax sensitive and language independent detection of code clones [Текст] / K. Maeda // World Academy of Science, Engineering and Technology 60. – 2009. – С. 350-354.

15. **Neill** Antipatterns in Systems Engineering: An Opening Trio [Текст] / Neill, J. Colin // INCOSE International Symposium. – 2012. – Vol. 22, no. 1. – С. 1233-1245. – ISSN 2334-5837.

16. **Pantel, P.** Discovering word senses from text. In Proceedings of the Eighth [Текст] / P. Pantel, D. Lin // ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. – 2002. – С. 613-619.

17. **Roy, C.K.** An empirical study of function clones in open source software systems [Текст] / C.K. Roy, J.R. Cordy // 15th Working Conference on Reverse Engineering (WCRE). – 2008. – С. 81-90.

18. **Wettel, R.** Archeology of code duplication: Recovering duplication chains from small duplication fragments [Текст] / R. Wettel, R. Marinescu / 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. – 2005.

19. **Brown**, **William J.** AntiPatterns and Patterns in Software Configuration Management. [Текст] / William J. Brown, W. Hays McCormick, W. Scott Thomas. // Wiley. – 1999. – ISBN 978-0-471-32929-9.

20. Антипаттерн, URL: https://ru.wikipedia.org/wiki/Антипаттерн

21. Векторная модель,
URL: https://ru.wikipedia.org/wiki/Векторная_модель

22. Документация по Visual Studio 2017,
URL: https://msdn.microsoft.com/ru-ru/library/hh205279.aspx

23. Программирование методом копирования-вставки,
URL: https://ru.wikipedia.org/wiki/Программирование_методом_копирования-вставки

24.Шаблон проектирования,
URL: https://ru.wikipedia.org/wiki/Шаблон_проектирования

25. CloneDr, URL: http://www.semdesigns.com/products/clone/

26. PMD, URL: https://pmd.github.io/